TRON Association

JTRON2.0 SPECIFICATION

September 14, 1999
Ver2.00.00

Java(tm) Technology
on ITRON-specification OS Technical Committee
ITRON Committe, TRON ASSOCATION

Editor: Yukikazu Nakamoto
Assistant Editor: Kazutoshi Usui
Page Layout Design: Kazutoshi Usui
Special thanks to
Wendy Fong
Natsuko Noda
Yoshiharu Asakura

Preface

An embedded system, in which a real-time OS(RTOS) has been used, is one of a strongly potential field for the application of Java technologies. Especially in Japan, the ITRON specification RTOS has been standardized to be used in many of the embedded systems. In applying Java technology to embedded systems, a hybrid approach is very promising;Java runtime environment is implemented on a real-time OS; parts of the application requiring real-time properties and parts of application requiring GUI features, extension of features, and replaced programs for new features should be implemented on Java runtime environment. In this case, the standardization of interfaces between the real-time task and Java program to encourage distribution of the programs are indispensable, from the view of the system development and the portability and reusability of real-time programs and Java programs. This specification provides for the interface specification between the real-time programs and the Java programs.

ITRON special committee
Java Technology on ITRON-specification OS technical committee
September 14, 1999

Note

- The copyright of this document belongs to the TRON Association.
- The authorization of the TRON Association is required in use of or copying part of the content of this document.
- The content of this specification may change without notice due to the future improvement.
- This document is based on Java Development Kit Version 1.1, not Java 2 platform.
- For the questions about this specification, ask:

  Tron Association
  Katsuta Building 5F, 1-3-39 Mita, Minato-ku, Tokyo 108-0073
  e-mail:jtron-question@itron.gr.jp

Remarks

- All the trademarks and logotypes of Java are trademarks of Sun Microsystems, Inc. in U.S.A. and other countries.
- Sun and Sun Microsystems are the trademarks of Sun Microsystems, Inc. in U.S.A. and the other countries.
- ITRON is the abbreviation of the "Industrial TRON", in which TRON is the abbreviation of the "The Real Operating System Nucleus".

Java Technology on ITRON-specification OS technical committee

Nobutaka Amano (formerly from Tron Association)

Katsuhiko Ishida (Hitachi, Ltd.)

Shoji Ueda (Metrowerks)

Kazutoshi Usui (NEC Corp.)

Tetsuo Oe (Oki Electric Industry Co., Ltd.)

Kenji Okazaki (Mentor Graphics Japan)

Masaya Kato (Toshiba Corp.)

Tomihisa Kamata (Access Corp.)

Tatsuya Kamei (Mitsubishi Electric Corp.)

Kenji Kudo (Fujitsu Device Inc.)

Yasuhiro Kobayashi (Fujitsu Ltd.)

Tetsu Shibashita (Mentor Graphics Japan)

Hiroyuki Suzuki (Access Corp.)

Tatsuya Koretsu (formerly from The University of Tokyo)

Hiroaki Takada (Toyohashi University of Technology)

Shuji Takanashi (Toshiba Corp.)

Toru Takeuchi (Tron Association)

Yukio Tada (Yamaha Corp.)

Noriaki Tanaka (Denso Create Inc.)

Kiichiro Tamaru (Toshiba Corp.)

Kenichi Nakamura (Nihon Cygnus Solutions)

Yukikazu Nakamoto (NEC Corp.; Manager)

Takeshi Narita (Toshiba Information Systems)

Shoichi Hachiya (Applix Coropration)

Seiji Hayashida (Toshiba Corp.)

Makoto Hirayama (Hewlett-Packard Company)

Tetsuo Miyauchi (NEC Microcomputer Technology, Ltd.)

Hiroyuki Muraki (Mitsubishi Electric Semiconductor Systems Corp.)

Takahiro Muranaka (Mitsubishi Electric Corp.)

Akihiro Yoshida (Applix Coropration)

Hiroyuki Watanabe (Seiko Instruments Inc.)

Contents

CONTENTS

List of Figures

List of Tables

Reference

[1] Tron Project, JTRON Specification, Dec. 1997.

[2] μITRON3.0 Standard Handbook " edited and published by Tron Association, Personal Media, 1997.

[3] JavaSoft, "Java Native Interface Specification Release 1.1", May,1997.

[4] J.Gosling, B. Joy and G. Steele, "The Java Language Specification", Addison-Wesley, 1996.

[5] Erich Gamma, Richard Helm, Ralph E. Johnson, John M. Vlissides, translated by Shinichi Honida, "Design Patterns: Abstraction and Reuse of Object-Oriented Design", Softbank, 1995.

[6] Toyokazu Tomatsu, "Java Program Design", Softbank, 1997.

Chapter 1

OVERVIEW

1.1 General

An embedded system, in which a real-time OS(RTOS) has been used, is one of a strongly potential field for the application of Java technologies. Especially in Japan, the ITRON specification RTOS has been standardized to be used in many of the embedded systems. In applying Java technology to embedded systems, a hybrid approach is very promising; Java runtime environment is implemented on a real-time OS; parts of the application requiring real-time properties, for instance handling multi-media stream, are implemented on the real-time OS; and parts of application requiring GUI features, extension of features, and eplaced programs for new features should be implemented on Java runtime environment. For example, device drivers and interrupt handlers should be written in C/C++ and executed on the real-time OS. In this case, the standardization of interface between the real-time task and Java program to encourage distribution of the programs are indispensable from the view of the system development and the portability and reusability of real-time programs and Java programs. This specification provides for the interface specification between the real-time programs and the Java programs.

The following are two interfaces between a real-time task and the Java program:

(1) Definition of the relation between a Java thread and a real-time task.
The Java thread is mapped to a real-time task in a one-to-one way. This mapping rule is provided.

(2) Definition of the cooperative computation for the Java program and the real-time taskEN:COOP

The following types are considered for the above:

Type 1 : Attach class
Allows use of the ITRON kernel system call in the Java program (corresponds with the JTRON specification [1]).

Type 2 : Shared object interface
The Java program and the real-time task communicate through shared objects.

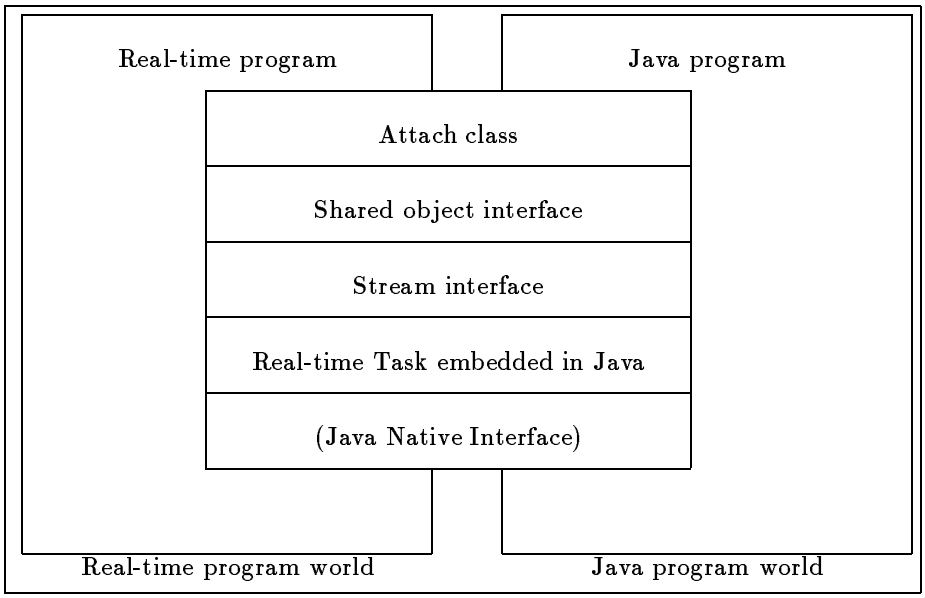


Figure 1.1: Cooperation of Java program and real-time program

Type 3 : Stream interface
The Java program and the real-time task communicate through streams.

Type 1 and 2 are regarded as tightly-coupled multi-processor system and Type 3 as loosely-coupled multi-processor system. The following approach is also available:

Type 4 Brings the Java programming in the real-time task.
This is done by calling the Java API from the real-time program using the JNI (Java Native method Interface[3]).

The contents defined in this specification are applicable not only among ITRON specification RTOS but also in a real-time OS provided from other vendors.

1.2 Overall Rules (ITRON Kernels)

1.2.1 Naming rules

In general, naming rules follow the naming rules in the ITRON specification. JTI (JTron Interface) is added as a prefix.

Macro name :	JTI_ZZZ
Type name :	T_JTI_XXX
Function name :	jti_XXX_YYY: XXX indicates an operation, and the YYY indicates the target object of the ope

1.2.2 Static API and dynamic API

For each API which creates objects, APIs(called static API) described in the configuration file are sent to a target system to create an object based on the configuration information when initializing the system. The static APIs can be distict from normal APIs(called dynamic API) by describing the API names in capital letters.

1.2.3 Return values and error codes of API

A return value of each API follows the ITRON specification conventions. The return value will be a negative value error code, in case of an error, and will be 0 or a positive value when executed normally. The meaning of the return value when executed normally is defined for each API.

Error codes consist of main error code, sub error code,and implementation-dependent error code. Main error codes, sub error codes, and implementation-dependent error codes are all negative values, and the error codes which combined some of these error codes are also negative. In JTRON2.0 specification, both of the main error codes and sub error codes are 8 bits and implementation-dependent error codes are 0 bit or bigger.

The following macros are provided to hide the implementation of the error codes.

JTI_MAINERCD(ercd)	Main error codes
JTI_SUBERCD(ercd)	Sub error codes
JTI_IMPLERCD(ercd)	Implementation-dependent error codes

Note: The above macro names will be replaced when the specification for overall ITRON specification is defined.

The mnemonics, values, and meanings of the main error codes must be standardized to be the same as the error codes in the ITRON kernel specification. However, the error code (E_CLS)which is not defined in the ITRON kernel specification must be defined additionally.

Sub error codes are expected to be used in the following way: If an exception occurs when executing the API which accesses the Java object from the real-time task, the main error code of the ITRON API error code is E_OBJ and a Java exception code is assigned to the sub error code. The Sub error codes are expected to be used to return useful information for debugging.

The implementation-dependent error code is defined according to the vendor implementation of JTRON specification 2.0.

In this specification, only the main error codes are defined as error codes each API returns.

The error codes below are not described for each API, but all the APIs may return those error codes. Some APIs may return the error code in the following: (but which API returns which error is implementation-dependent.)

E_SYS	System error
E_NOMEM	Not enough memory
E_NOSPT	Function not supported
E_MACV	Memory access violation

1.2.4 Waiting status and time-out

When the execution of the program is waiting until a certain event takes place in the real-time task, the status is called "waiting" or "entering the waiting status", and in the Java thread, it is called "blocked".

ITRON APIs that may enter waiting status provides a timeout functionality.

THe time-out functionality is to return from the API by canceling the process when the process is not complete after a certain period of time (in this case, an E_TMOUT error is returned from the API). For this reason, the status of the object does not change by calling the API in case of the time-out. The exception is when the object status cannot be returned to a status before calling the APIs in canceling the process due to the function of the API.

Polling is the time-out process which sets the time-out time to 0.

When a program calling an API enters a waiting status, the process by the API is said to be pending.

In the API description in this specification, the behavior without time-out (permanent waiting) is described. Even if "waiting status" is found in the API function explanation, the waiting status is released after a specified time and the status returns from API with E_TMOUT as the return value if the time-out has been specified.

The time-out value indicates the time-out time (millisecond is recommended), for positive values, a polling for
TMO_POL (= 0) and permanent waiting for TMO_FEVR(= -1).

1.2.5 Relation between API and tasks

The APIs in this specification act in the same way even if called from a different task, if the parameters are the same, which means there is no resource to be assigned to the task by the APIs in this specification.

When the task A calls the API in this specification and enters in the waiting status, and another task B wakes up the task A by issuing rel_wai system call, an E_RLWAI error is returned from the API from the task A. If a ter_tsk was issued in the same situation, the behavior is implementation-dependent.

1.3 Common Definition

1.3.1 Header files

Header files are described as :"jti_XXX.h"

Header file used for the type 1 : "jti_attach.h"
Header file used for the type 2 : "jti_shared.h"
Header file used for the type 3 : "jti_stream.h"

1.3.2 Data structure / Data type

(1) For shared object interface

JNO Integer type, length is implementation-dependent
ER Integer type, 16 bits or bigger for JTRON

(2) For stream interface

```
typedef struct t_jti_cstm {
    VP     exinf;     /* Extension information */
    ATR    stmatr;    /* Stream attribute */
    VP     wbuf;      /* Head of the sending buffer */
    INT    wbufsz;    /* Size of the sending buffer */
    VP     rbuf;      /* Head of the receiving buffer */
    INT    rbufsz;    /* Size of the receiving buffer */
/* Other implementation-dependent fields can also be added. */
} T_JTI_CSTM;

typedef struct t_jti_rstm {
    VP     exinf;     /* Extension information */
    INT    wrisz;     /* Data length which can be sent without waiting */
    INT    reasz;     /* Data length which can be received without waiting */
/* Other implementation-dependent fields can also be added. */
} T_JTI_RSTM;
```

1.3.3 Constants

(1) General

NADR	−1	Invalid address

(2) API function codes
(Omission)

(3) Main error codes

E_OK	0	Normal termination
E_SYS	−5	System error
E_NOMEM	−10	Not enough memory
E_NOSPT	−17	Function not supported
E_RSATR	−24	Reserved attribute
E_PAR	−33	Parameter error
E_ID	−35	Illegal ID number
E_NOEXS	−52	Object not created
E_OBJ	−63	Object status error
E_MACV	−65	Memory access violation
E_DLT	−81	Deletion of the waiting status
E_RLWAI	−86	Cancellation of the process, compulsory cancellation of the waiting status
E_CLS	−87	Disconnected

(4) BOOL values

TRUE	1	True
FALSE	0	False

(5) Time-out specification

TMO_POL	0	Polling
TMO_FEVR	−1	Permanent waiting

(6) Java thread / Real-time task priority specification

JTI_DFL_HPR	Default highest priority value for real-time task implementing JRE. The value is implementatio

(7) For stream interface

JTI_MAIN_STREAM	1	Main stream ID
TA_WRITE	0x01	Stream attribute. Enables sending.
TA_READ	0x02	Stream attribute. Enables receiving.

(8) Error-obtaining macros

JTI_MAINERCD(ercd)	Main error code
JTI_SUBERCD(ercd)	Sub error code
JTI_IMPLERCD(ercd)	Implementation-dependent error code

1.4 Overall Rules (Java)

1.4.1 JTRON standard Java package structure

The Java class package names which provide JTRON2.0 specification should be unique if the package specification is the same. Following the Java language specification, the package name starts with the Internet domain name (XXX) followed by a name (YYY) used for the identification for management (Table1.1).

Table 1.1: JTRON standard Java package names

Type	Package name format	JTRON standard Java package name
Package used for type 1:	XXX.jtron.attach.YYY	jp.gr.itron.jtron.attach.YYY
Package used for type 2:	XXX.jtron.shared.YYY	jp.gr.itron.jtron.shared.YYY
Package used for type 3:	XXX.jtron.stream.YYY	jp.gr.itron.jtron.stream.YYY

When a vendor expands functionality of the package, the vendor must add a name according to the JTRON standard Java package names, which means the domain name of the vendor comes after XXX. Therefore, the package structure can remain same for convenience of programmer. Vendors are not allowed to give the same name for the different functions. If the functions are different, the vendor must change the name or create an unique package name for the vendor.

1.4.2 JTRON standard Java class structure

In the class definition, method names or variable names which are shown to programmers are public and those which depend on vendors are not public.

In this specification, the overriding methods among the methods defined in the super class are not described. Vendors must appropriately set the overriding as required.

Examples

- Object#toString()
- Throwable#getMessage()

1.4.3 Java system property

The following system properties are provided as standard.

jtron.version :
The version number of the JTRON specification provided (complies with the convention for the μITRON version number).

jtron.type :

Type number expressed by the combination of more than one of the following alphanumeric characters.

0:	Attach class
1:	Shared object interface
2:	Stream interface
3–9,A–Z:	Reserved for future use

jtron.vendor :

Vendor name (can be set to vendor's convenience).

These property values can be obtained through the getProperty method in the jp.gr.itron.jtron.JtiSystem class.

1.5 Operating Rules

(1) Maintenance
The specification must be reviewed in the first half of the year 1999 when companies are likely to implement the specification and evaluate it. The consistency with the μITRON4.0 specification must be considered at the same time.

(2) Compliance
Those which implement any of the specification for Type 1, Type 2, or Type 3 (excluding the extended specification) are allowed to be announced as "JTRON2.0-compliant". A label "Standard" in API specification stands for a mandatory API, "Extension" for an optional API.

(3) Registration and approval system
We employ the registration system to certify the products which comply with the specification. However, we do not give any official approval for whether or not the product complies with the specification.

Chapter 2

MAPPING OF JAVA THREAD AND REAL-TIME TASKS

2.1 General

In JTRON2.0 specification, one Java thread is mapped to one real-time task in one-to-one way. The following mapping rules are provided:

(1) Defines the relationship of the priority between the Java thread and the real-time task. This means that the highest priority among all the priorities of the real-time task which implements JRE (Java Run-time Environment) can be defined.

[**Rationale**]

The following rules were also considered in providing the priority mapping between the Java thread and the real-time task.

- Defines an API which can set and refer to the priority mapping table for the Java thread and the real-time task.
- Defines an API which can define the highest priority and the lowest priority of the real time task implementing the JTRON2.0 interface(or the highest priority and the lowest priority of the real-time task, which can be used in the real-time programs because the JRE use real-time task priorities, so that priorities, which can be used in the real-time programs are limited).

These methods were not employed due to the reasons below:

- The real-time task concerns only the highest priority of the JTRON2.0 interface implementation and not the lowest priority.
- The priority available in the real-time task should be obtainable statically.

2.2 ITRON API

API Name	Function	Type
jti_set_hpr	Sets the highest priority of the real-time task implementing the JRE	Standard
jti_get_hpr	Gets the real-time task priority from the Java thread priority	Standard
jti_get_lpr	Gets the lowest priority of the real-time task implementing the JRE	Standard

Standard

jti_set_hpr

Sets the highest priority of the real-time task implementing the JRE

[**C language API**]

```
void jti_set_hpr(hijpr);
```

[**Static API**]

```
JTI_SET_HPR(hijpr)
```

[**Parameters**]

PRI *hijpr* Real-time task priority

[[**Return value**]

None

[**API function**]

Sets a value of the highest priority of the real-time task implementing the JRE to *hijpr*. **JTI_DFL_HPR** has been set as an initial value in case that this API is not executed.

[**Note**]

Even if the value of the highest priority is changed dynamically, the priority of the Java thread already in execution will not change.

Standard

jti_get_hpr

Gets the real-time task priority from the Java thread priority

[**C language API**]

```
PRI pri = jti_get_hpr(hijpr, jpr);
```

[**Static API**]

```
PRI pri = JTI_GET_HPR(hijpr, jpr);
```

[**Parameters**]

PRI	*hijpr*	the highest priority of the real-time task implementing the JRE
INT	*jpr*	Java thread priority

[**Return value**]

PRI	*pri*	Real-time task priority

[**API function**]

Gets a priority of the Java thread *jpr* in the ITRON kernel based on the highest priority value *hijpr* of the real-time task implementing the Java thread, and returns the priority. The static API is implementation-dependent including if the API is provided or not.

Standard

jti_get_lpr

Finds the lowest priority of the real-time task implementing the JRE

[**C language API**]

```
PRI pri = jti_get_lpr(hijpr);
```

[**Static API**]

```
PRI pri = JTI_GET_HPR(hijpr);
```

[**Parameters**]

PRI *hijpr* the highest priority of the real-time task implementing the JRE

[**Return value**]

PRI *pri* The lowest priority of the real-time task implementing the JRE

[**API function**]

Retrieves the lowest priority value of real-time task implementing the JRE based on the highest priority value *hijpr* of the real-time task implementing the Java thread, and returns the priority. The static API is implementation-dependent including if the API is provided or not.

2.3 Java API

2.3.1 Package structure

The classes which manage and control the overall JTRON system are collected in the jp.gr.itron.jtron package.This package consists of the following class.

Class: JtiSystem

2.3.2 Class jp.gr.itron.jtron.JtiSystem

```
java.lang.Object
   |
   +--- jp.gr.itron.jtron.JtiSystem
```

---

public JtiSystem

Manages the information such as property concerning the JTRON interface specification.

□ **Constructor**

protected JtiSystem()

□ **Methods**

public static JtiSystem getJtiSystem()
Obtains an object of the JtiSystem class. By calling this method, the JTRON mechanism on the Java side (mechanism in which the ITRON real-time task controls the Java resource) is available.

public String getProperty(String key)
Obtains the JTRON system property indicated by the specified key.

public String getProperty(String key, String default)
Obtains the JTRON system property indicated by the specified key. Returns the default if the property specified by the key cannot be found.

public Properties getProperties()
Obtains the JTRON system property. The following properties are defined as standard:

jtron.version :
A version number of the JTRON specification provided (complies with the convention for the μITRON version number)

jtron.type :

Type number expressed by the combination of more than one of the following alphanumeric characters.

0:	Attach class
1:	Shared object interface
2:	Stream interface
3–9,A–Z:	Reserved for future usage

jtron.vendor :

Vendor name (can be set to vendor's convenience)

Chapter 3

ATTACH CLASS

Plese refer to the JTRON specification found in the reference [1] JTRON1 specification. For attach classes, the JTRON1 specification is applicable to up to μITRON3.0[2]of the real-time OS specification. The attach class specification will be updated corressponding to the μITRON4.0 specification.

Chapter 4

SHARED OBJECT INTERFACE

4.1 General

The shared object interface provides a communication means by exchanging data between Java threads and real-time tasks (Figure 4.1).

In the Java thread, an object for sharing is registered in the shared object manager which controls the exchange with the real-time task. In the real-time task, a head address of the registered shared object is obtained. Data is exchanged between the Java thread and the real-time task by using this shared object. A lock mechanism is provided to keep the consistency of the shared object.

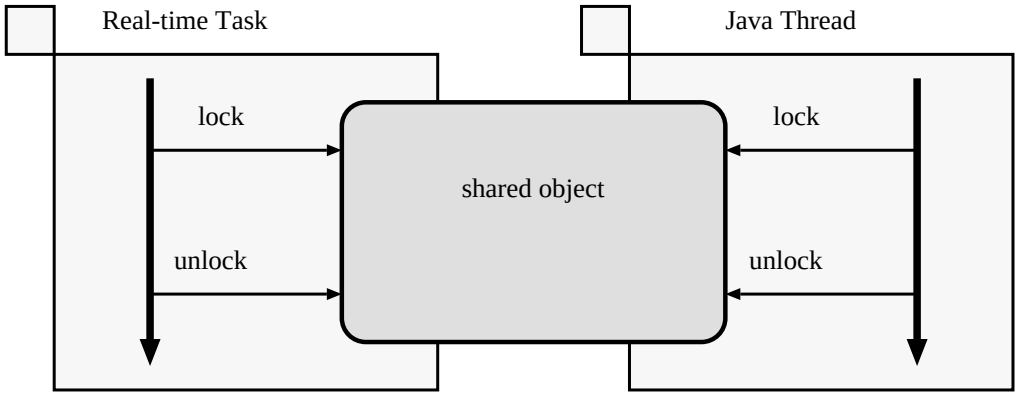


Figure 4.1: Shared object

SharedObjectManager class is provided as a class for exchanging with the real-time task. All the other

classes exchange data with the real-time task through the **SharedObjectManager**. By inheriting the **SharedObject** class provided as a Java class library or by implementing the Sharable interface, a Java object shared with the real-time task side is created. When creating this shared object, a name is given to register in the **SharedObjectManager**. When accessing the shared object, locking or unlocking (**lock** method and **unlock** method of the **SharedObject** in the Java program and **loc_shm** and **unl_shm** API in the ITRON task) must be performed for the mutual exclusion. From the real-time task, a shared object is locked first and a head address of the shared object will be taken out to access the shared object. The unlocking will be done after completing the access. The address of the shared object after being unlocked is not guaranteed since the sharing may be finished.

Among the **Thread** classes and **ThreadGroup** classes of the Java, execution control methods which involves the status transition can be called from the real-time tasks an extension specification. This is because the control of the Java thread from the real-time task is required when communicating with the shared object.

Semantics of lock

Shown below is the status transition of the shared object when the Java thread or the real-time task executed a locking or an unlocking in the shared status, which also can be divided into locked status and not locked status, and in the unshared status.

"Same lock owner" stands for the case in which a real-time task or a Java thread, that previously executed the lock operation and the thread or the real-time task, that is to execute the current operation are identical. "Different lock owner (Java thread)" stands for the case in which the thread which previously executed the lock operation and the thread or the real-time task, that is to execute the current operation are different. "Different lock owner (Real-time task)" stands for the case in which the task which previously executed the lock operation and the thread or the real-time task which is to execute the current operation are different. These are shown in the table 4.1.

If a ThreadDeath exception takes place through the **stop** method by another thread B after the Java thread A executed the **lock** method, the thread A executes the **unlock** method in the finally sentence which processes this exception to unlock the object.

The waiting order of the shared object in locked status is implementation-dependent.

Relation with the garbage collection (GC)

(1) The shared objects after an **share** method was issued (**share**method is normally issued in the constructor) are not the target of the GC.

(2) The shared objects after an **unshare** method was issued are the target of the GC and the sharable objects do not automatically disappear. You have to pay attention since the shared objects cannot be the target of the GC unless you clearly execute the **unshare** method.

Table 4.1: Shared object lock status transition

operation		lock				unlock
		not locked	locked			
			Same lock owner	Different lock owner (Java thread)	Different lock owner (Real-time task)	
Java method	lock	OK (lock)	OK (no effect)	Blocked	Blocked	Exception
	unlock	OK (no effect)	OK (Unlock)	Exception	Exception	Exception
	forceUnlock *1	OK (no effect)	OK (Unlock)	OK (Unlock)	OK (no effect)	Exception
	unshare *2	OK	*3	*4	*4	Exception
ITRON API	jti_loc_obj	OK (lock)	OK (no effect)	Waiting status	Waiting status	E_OBJ error
	jti_unl_obj	OK (no effect)	OK (Unlock)	E_OBJ error	E_OBJ error	E_OBJ error
	jti_funl_obj *5	OK (no effect)	OK (Unlock)	OK (Unlock)	OK (Unlock)	E_OBJ error

***1** The **forceUnlock** method unlocks an arbitrary Java thread by force. This method does not unlock a lock of the real-time task. This method is provided because another Java thread can perform the unlocking by force in case that the Java thread which locked the shared data is dead.

***2** This method ends the sharing of the shared object and only Java thread can end the sharing. In the **unshare** method, in order to finish the sharing the shared object is locked first and then performs to finish the sharing, and at last unlocks the object. Therefore, if the object to be ended the sharing has been already locked by the Java thread or the real-time task, the thread which issued the **unshare** method will be blocked until the object is unlocked. This is to safely end the sharing after the access of the real-time task and the Java thread since the **unshare**method is executed asynchronously.

***3** Ends the sharing of the object after unlocking.

***4** Enters the waiting status until other threads or tasks unlock the object.

***5** If the real-time task intends to unlock the shared object locked by the real-time task or the Java thread by force (**jti_funl_obj**), the object will be unlocked.

[**Supplementary explanation**]

- **Expected execution orders**

  The execution is expected to be done in the order shown in the figure 4.2.

Real-time task	**Java program**
	1: Registers the **ShareObject** by giving a name
2: The **jti_get_obj** obtains the shared object identification number by using the name	
3: The **jti_loc_obj** locks the shared object	
4: The **jti_get_mem** obtains an address of the shared object	
5: Make an access by using the address	
6: The **jti_unl_obj** unlocks the shared object	
7: Notifys the end of accessing the shared object to the Java thread	
	8: Executes the **lock** method for the **ShareObject**
	9: Executes the **unlock** method for the **SharedObject**

Figure 4.2: Expected operation orders

- **Assumptions for the shared Java objects**

  The memory location in the Java object (Endian, alignment, padding, etc.) is found in C structures by the use of tools such as javah command. This assumption must be reviewed as soon as possible, since existence of a function to obtain the memory location information other than JNI is not assured yet.

  Any system call can be executed between the execution of **jti_loc_obj** and **jti_unl_obj**, therefore, the task will enter in any status. It is preferred that the task being locked must remain in the run/ready status.

  A try_lock function (locks the object when the object can be locked and issues an error or an exception when the object cannot be locked) can be realized by specifying 0 to the waiting time when being locked.

4.2 ITRON API

4.2.1 ITRON API for accessing shared objects

API Name	Function	Type
jti_get_obj	Finds the shared object identification number by using names	Standard
jti_get_mem	Returns a head address in the memory region corresponding the specified shared object (Whose class name is **Sharable**)	Standard
jti_loc_obj	Locks the specified Java object	Standard
jti_unl_obj	Unlocks the specified shared object	Standard
jti_funl_obj	Unlocks the specified shared object by force	Standard

Standard

jti_get_obj

Finds the shared object identification number by using names

[**C language API**]

```
ER ercd = jti_get_obj(char *objnm, JNO *p_objno);
```

[**Parameters**]

char	**objnm*	Shared object name
JNO	**p_objno*	Shared object identification number

[**Return value**]

ER	*ercd*	Error code

[**Error code**]

E_OK	Normal termination
E_OBJ	No shared object corresponding the *obknm* exists
E_PAR	Wrong parameter (*objnm* is a NULL pointer)

[**API function**]

Returns the Java shared object identification number corresponding the *objnm* to the region *p_objno* specifies. The API regards the *objnm* character string as an UTF-8 character string and returns the identification number of the Java object which has the identical name. If no Java object corresponding the *objnm* is found, an **E_OBJ** is returned. If the *objnm* is a NULL pointer, an **E_PAR** is returned. The implementation may limit the *objnm* to the ASCII character strings.

Standard

jti_get_mem

Returns the head pointer of the specified shared object (Whose class name is Sharable)

[C language API]

```
ER ercd = jti_get_mem(JNO objno, VP* p_addr);
```

[Parameters]

JNO	*objno*	Shared object identification number
VP*	*p_addr*	The pointer for the region to store the head address of the shared object

[Return value]

ER	*ercd*	Error code

[Error codes]

E_OK	Normal termination
E_PAR	Wrong parameter
E_OBJ	No object exists

[API function]

Returns a head address of the shared object specified by *obj* to the region specified by the *p_addr*. A programmer has to access the address stored in the region specified by the *p_addr* by casting the type definition corresponding the Java object. Refer to the JNI specification [3] for type correspondence between Java programming language and C language.

Standard

jti_loc_obj

Locks the specified Java object

[**C language API**]

```
ER ercd = jti_loc_obj(JNO objno, TMO tmout);
```

[**Parameters**]

JNO	*objno*	Shared object identification number
TMO	*tmout*	Time-out time

[**Return value**]

ER	*ercd*	Error code

[**Error codes**]

E_OK	Normal termination
E_PAR	Wrong parameter
E_OBJ	No shared object exists
E_TMOUT	Time-out takes place
E_RLWAI	Waiting status released by force
E_DLT	Sharing released

[**API function**]

Locks the shared object specified with the *objno*. The following are cases where the object has been previously locked.

(1) The object has been locked by the **lock** method of the class **SharedObject** on the Java thread.

(2) The object has been locked by the **loc_obj** on the different real-time task .

If the object has been previouly locked, the task enter will enter in the waiting status. In this waiting status, the object will be unlocked by the **unlock** method of the class **SharedObject** in the Java program or the **unl_obj** of the real-time task. If the object has been locked by the same real-time task, the API terminates normally without taking any action.

Standard

jti_unl_obj

Unlocks the specified shared object

[**C language API**]

```
ER ercd = jti_unl_obj(JNO objno);
```

[**Parameters**]

JNO *objno* Shared object identification number

[**Return value**]

ER *ercd* Error code

[**Error codes**]

E_OK Normal termination
E_PAR Wrong parameter (Illegal *objno*)
E_OBJ No object exists, or intended to unlock the object locked by a different task

[**API function**]

Unlocks the shared object locked by the same real-time task specified by the *objno*. If the shared object specified by the *objno* has been locked by a different real-time task or the Java thread, an **E_OBJ** error is returned.

Standard

jti_funl_obj

Unlocks the specified shared object by force

[**C language API**]

```
ER ercd = jti_funl_obj(JNO objno);
```

[**Parameters**]

JNO *objno* Shared object identification number

[**Return value**]

ER *ercd* Error code

[**Error codes**]

E_OK Normal termination
E_PAR Wrong parameter (Illegal objno)
E_OBJ No object exists

[**API function**]

Unlocks the shared object specified by the *objno* by force regardless of the Java thread or the real-time task which locked the object.

4.2.2 ITRON API for operating the Java thread

For the methods of Java **Thread** classes which appear in the following API explanation, refer to [4].

API Name	Function	Type
jti_get_thr	Gets the thread identification number by using names	Extension
jti_isa_thr	Calls the isAlive method in the Java Thread class	Extension
jti_int_thr	Calls the interrupt method in the Java Thread class	Extension
jti_isi_thr	Calls the isInterrupted method in the Java Thread class	Extension
jti_sus_thr	Calls the suspend method in the Java Thread class	Extension
jti_rsm_thr	Calls the resume method in the Java Thread class	Extension
jti_sta_thr	Calls the start method in the Java Thread class	Extension
jti_thr_stp	Calls the stop method in the Java Thread class	Extension
jti_get_jpr	Calls the getPriority method in the Java Thread class	Extension
jti_set_jpr	Calls the setPriority method in the Java Thread class	Extension
jti_des_thr	Calls the destroy method in the Java Thread class	Extension

Extension

jti_get_thr

Gets the thread identification number by using names

[**C language API**]

```
ER ercd = jti_get_thr(char *thrnm, JNO *p_thrno);
```

[**Parameters**]

char	**thrnm*	Java thread name
JNO	**p_thrno*	Java thread identification number

[**Return value**]

ER	*ercd*	Error code

[**Error codes**]

E_OK	Normal termination
E_OBJ	No thread exists
E_PAR	Wrong parameter (*thrnm* is a NULL pointer)

[**API function**]

Returns the Java thread identification number corresponding the *thrnm* to the region *p_thrno* specifies. The API regards the *thrnm* character string as an UTF-8 character string and returns the identification number of the Java thread which has the identical name. If no Java object corresponding the *thrnm* is found, an **E_OBJ** is returned. If the *thrnm* is a NULL pointer, an **E_PAR** is returned. The implementation may limit the *thrnm* to the ASCII character strings.

Extension

jti_isa_thr

Calls the isAlive method in the Java Thread class

[**C language API**]

```
ER_BOOL ercd = jti_isa_thr(JNO thrno);
```

[**Parameters**]

JNO *thrno* Java thread identification number

[**Return value**]

ER_BOOL *ercd* Return value of the method or an error code

[**Error codes**]

TRUE True
FALSE False
E_PAR Wrong parameter (Illegal *thrno*)

[**API function**]

Calls the **isAlive** method in the Thread class for the Java thread specified by the *thrno* and returns the result.

Extension

jti_int_thr

Calls the interrupt method in the Java Thread class

[**C language API**]

```
ER ercd = jti_int_thr(JNO thrno);
```

[**Parameters**]

JNO *thrno* Java thread identification number

[**Return value**]

ER *ercd* Error code

[**Error codes**]

E_OK Normal termination
E_PAR Wrong parameter (Illegal *thrno*)

[**API function**]

Calls the **interrupt** method in the Thread class for the Java thread specified by the *thrno*.

Extension

jti_isi_thr

Calls the isInterrupted method in the Java Thread class

[**C language API**]

```
ER_BOOL ercd = jti_isi_thr(JNO thrno);
```

[**Parameters**]

JNO *thrno* Java thread identification number

[**Return value**]

ER_BOOL *ercd* Return value of the method or an error code

[**Error codes**]

TRUE True
FALSE False
E_PAR Wrong parameter (Illegal *thrno*)

[**API function**]

Calls the **isInterrupted** method in the Thread class for the Java thread specified by the *thrno* and returns the result.

Extension

jti_sus_thr

Calls the suspend method in the Java Thread class

[**C language API**]

```
ER ercd = jti_sus_thr(JNO thrno);
```

[**Parameters**]

JNO *thrno* Java thread identification number

[**Return value**]

ER *ercd* Error code

[**Error codes**]

E_OK Normal termination
E_PAR Wrong parameter (Illegal *thrno*)
E_OBJ A security exception took place while executing the Java method

[**API function**]

Calls the **suspend** method in the **Thread** class for the Java thread specified by the *thrno*. The condition in which the security exception took place depends on the implementation of the security manager.

Extension

jti_rsm_thr

Calls the resume method in the Java Thread class

[**C language API**]

```
ER ercd = jti_rsm_thr(JNO thrno);
```

[**Parameters**]

JNO *thrno* Java thread identification number

[**Return value**]

ER *ercd* Error code

[**Error codes**]

E_OK Normal termination
E_PAR Wrong parameter (Illegal thrno)
E_OBJ A security exception took place while executing the Java method

[**API function**]

Calls the **resume** method in the Thread class for the Java thread specified by the *thrno*. The condition in which the security exception took place depends on the implementation of the security manager.

Extension

jti_sta_thr

Calls the start method in the Java Thread class

[**C language API**]

```
ER ercd = jti_sta_thr(JNO thrno);
```

[**Parameters**]

JNO *thrno* Java thread identification number

[**Return value**]

ER *ercd* Error code

[**Error codes**]

E_OK Normal termination
E_PAR Wrong parameter (Illegal *thrno*)
E_OBJ A status violation took place while executing the Java method

[**API function**]

Calls the start method in the **Thread** class for the Java thread specified by the *thrno*.

Extension

jti_thr_stp

Calls the stop method in the Java Thread class

[**C language API**]

```
ER ercd = jti_thr_stp(JNO thrno);
```

[**Parameters**]

JNO *thrno* Java thread identification number

[**Return value**]

ER *ercd* Error code

[**Error codes**]

E_OK	Normal termination
E_PAR	Wrong parameter (Illegal *thrno*)
E_OBJ	A security exception or a NULL pointer exception took place while executing the Java method

[**API function**]

Calls the **stop** method in the **Thread** class for the Java thread specified by the *thrno*. The condition in which the security exception took place depends on the implementation of the security manager.

[**Supplementary explanation**]

Since the overriding method **stop(Throwable thrno)** is considered to be used infrequently, the method is excluded from the methods available to be called from the real-time task.

Extension

jti_get_jpr

Calls the getPriority method in the Java Thread class

[**C language API**]

```
ER ercd = jti_get_jpr(JNO thrno, INT *p_rslt);
```

[**Parameters**]

JNO	*thrno*	Java thread identification number
INT	**p_rslt*	Java thread priority

[**Return value**]

ER	*ercd*	Error code

[**Error codes**]

E_OK	Normal termination
E_PAR	Wrong parameter (Illegal *thrno*)

[**API function**]

Calls the **getPriority** method in the **Thread** class for the Java thread specified by the *thrno* and returns the result to the *p_rslt*.

[**Note**]

The priorities which can be obtained in this API are the priorities in the Java thread, not real-time task.

Extension

jti_set_jpr

Calls the setPriority method in the Java Thread class

[**C language API**]

```
ER ercd = jti_set_jpr(JNO thrno, INT newpri);
```

[**Parameters**]

JNO	*thrno*	Java thread identification number
INT	*newpri*	Java thread priority

[**Return value**]

ER	*ercd*	Error code

[**Error codes**]

E_OK	Normal termination
E_PAR	Wrong parameter (Illegal *thrno*)
E_OBJ	A security exception or a wrong argument exception took place while executing the Java method

[**API function**]

Calls the **setPriority** method in the **Thread** class for the Java thread specified by the *thrno*. The condition in which the security exception took place depends on the implementation of the security manager.

[**Note**]

The priorities which can be obtained in this API are the priorities in the Java thread, not real-time task.

Extension

jti_des_thr

Calls the destroy method in the Java Thread class

[**C language API**]

```
ER ercd = jti_des_thr(JNO thrno);
```

[**Parameters**]

JNO *thrno* Java thread identification number

[**Return value**]

ER *ercd* Error code

[**Error codes**]

E_OK Normal termination
E_PAR Wrong parameter (Illegal *thrno*)
E_OBJ A security exception took place while executing the Java method

[**API function**]

Calls the **destory** method in the **Thread** class for the Java thread specified by the *thrno*. The condition in which the security exception took place depends on the implementation of the security manager.

4.2.3 ITRON API for operating Java thread groups

For the methods of the Java ThreadGroup class in the following API explanation, refer to [4].

API Name	Function	Type
jti_get_tgr	Gets the Java thread group identification number by using names	Extension
jti_des_tgr	Calls the destroy method in the Java ThreadGroup class	Extension
jti_sus_tgr	Calls the suspend method in the Java ThreadGroup class	Extension
jti_rsm_tgr	Calls the resume method in the Java ThreadGroup class	Extension
jti_stp_tgr	Calls the stop method in the Java ThreadGroup class	Extension

Extension

jti_get_tgr

Gets the Java thread group identification number by using names

[**C language API**]

```
ER ercd = jti_get_tgr(char *tgrnm, JNO *p_tgrno);
```

[**Parameters**]

char	**tgrnm*	Java thread group name
JNO	**p_tgrno*	Java thread group identification number

[**Return value**]

ER	*ercd*	Error code

[**Error codes**]

E_OK	Normal termination
E_OBJ	No thread group exists
E_PAR	Wrong parameter (*tgrnm* is a NULL pointer)

[**API function**]

Returns the Java thread group identification number corresponding the *tgrnm* to the region *p_tgrno* specifies. The API regards the *tgrnm* character string as an UTF-8 character string and returns the identification number of the Java thread group which has the identical name. If no Java object corresponding the *tgrnm* is found, an **E_OBJ** is returned. If the *tgrnm* is a NULL pointer, an **E_PAR** is returned. The implementation may limit the *tgrnm* to the ASCII character strings.

Extension

jti_des_tgr

Calls the destroy method in the Java ThreadGroup class

[**C language API**]

```
ER ercd = jti_des_tgr(JNO tgrno);
```

[**Parameters**]

JNO *tgrno* Java thread group identification number

[**Return value**]

ER *ercd* Error code

[**Error codes**]

E_OK Normal termination
E_PAR Wrong parameter (Illegal *tgrno*)
E_OBJ A security exception or a status violation took place while executing the Java method

[**API function**]

Calls the **destroy** method in the **ThreadGroup** class for the Java thread group specified by the *tgrno*. The condition in which the security exception took place depends on the implementation of the security manager.

Extension

jti_sus_tgr

Calls the suspend method in the Java ThreadGroup class

[**C language API**]

```
ER ercd = jti_sus_tgr(JNO tgrno);
```

[**Parameters**]

JNO *tgrno* Java thread group identification number

[**Return value**]

ER *ercd* Error code

[**Error codes**]

E_OK Normal termination
E_PAR Wrong parameter (Illegal *tgrno*)
E_OBJ A security exception took place while executing the Java method

[**API function**]

Calls the **suspend** method in the **ThreadGroup** class for the Java thread group specified by the *tgrno*. The condition in which the security exception took place depends on the implementation of the security manager.

Extension

jti_rsm_tgr

Calls the resume method in the Java ThreadGroup class

[**C language API**]

```
ER ercd = jti_rsm_tgr(JNO tgrno);
```

[**Parameters**]

JNO *tgrno* Java thread group identification number

[**Return value**]

ER ercd
Error code

[**Error codes**]

E_OK Normal termination
E_PAR Wrong parameter (Illegal *tgrno*)
E_OBJ A security exception took place while executing the Java method

[**API function**]

Calls the **resume** method in the **ThreadGroup** class for the Java thread group specified by the *tgrno*. The condition in which the security exception took place depends on the implementation of the security manager.

Extension

jti_stp_tgr

Calls the stop method in the Java ThreadGroup class

[**C language API**]

```
ER ercd = jti_stp_tgr(JNO tgrno);
```

[**Parameters**]

JNO *tgrno* Java thread group identification number

[**Return value**]

ER *ercd* Error code

[**Error codes**]

E_OK Normal termination
E_PAR Wrong parameter (Illegal *tgrno*)
E_OBJ A security exception took place while executing the Java method

[**API function**]

Calls the **stop** method in the **ThreadGroup** class for the Java thread group specified by the *tgrno*. The condition in which the security exception took place depends on the implementation of the security manager.

4.3 Java API

4.3.1 Package structure

The classes providing the shared objects are collected in the jp.gr.itron.jtron.shared package. The package consists of the following interface, classes, and exception classes.

Interface: Sharable

Class: SharedObject,SharedObjectManager

Exception class : ShmException,ShmIllegalStateException,ShmTimeoutException

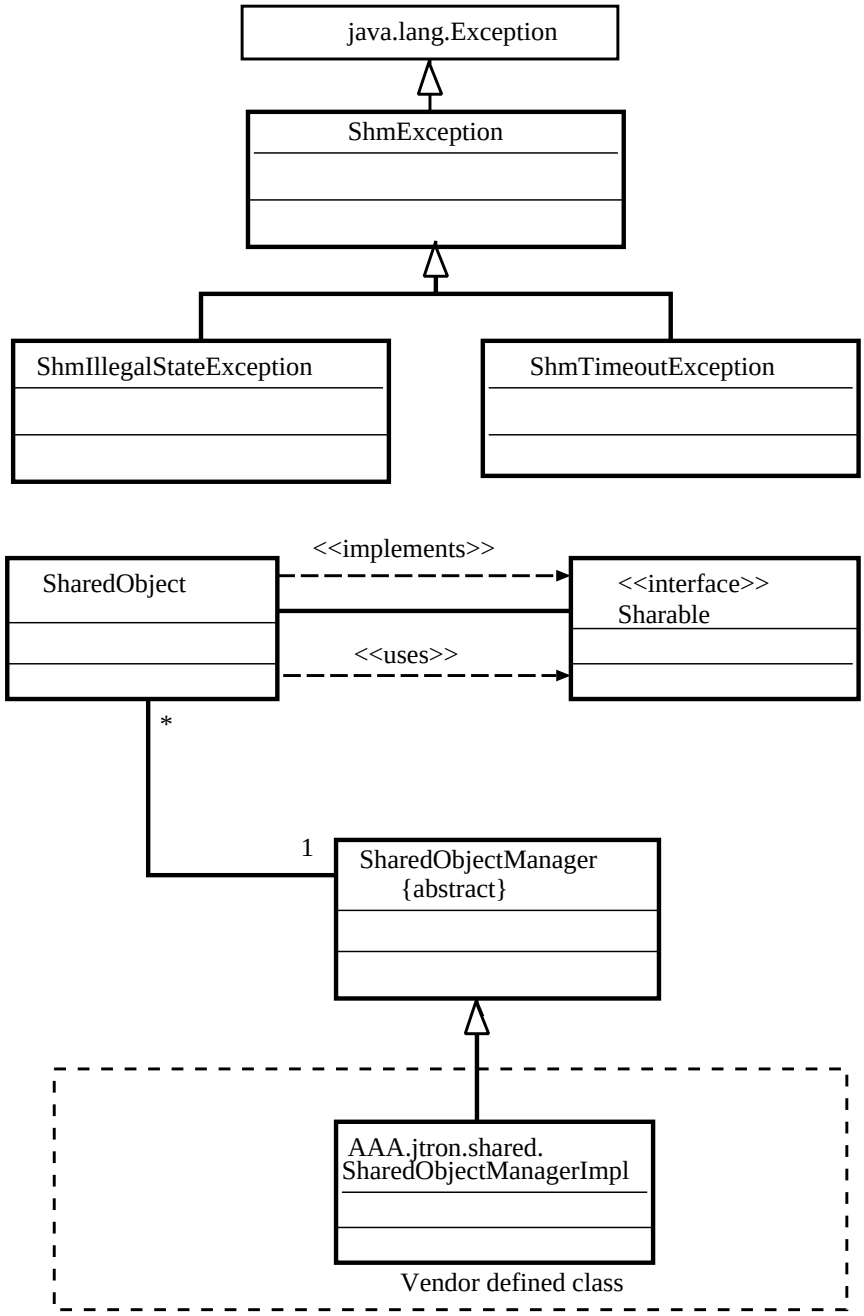


Figure 4.3: Shared package class structure

4.3.2 Interface jp.gr.itron.jtron.shared.Sharable

public interface Sharable

Provides the interface for the shared objects. The class of the object which is to be used as a shared object must implement this interface.

□ **Methods**

public abstract void lock()
Locks the object. No action takes place if the object has been locked by the same thread. If the object has been locked by a different thread or the real-time task, the method will be blocked until the object is unlocked.

public abstract void lock(int timeout) throws ShmTimeoutException
Locks the object. No action takes place if the object has been locked by the same thread. If the object has been locked by a different thread or the real-time task, the method will be blocked for the *timeout* (unit ms), *timeout* time, until the object is unlocked. If the time-out time has elapsed, a **ShmTimeoutException** will be thrown.

public abstract void unlock() throws ShmIllegalStateException
Unlocks the object locked by the same thread. No action takes place if the object has been unlocked.
Throws a **ShmIllegalStateException** for the object which has been locked by a different thread or the real-time task.

public abstract void forceUnlock()
Unlocks the object which has been locked by a thread by force. No action takes place for the object being locked by the real-time task.

public abstract void unshare() throws ShmIllegalStateException
Terminates the object sharing with the real-time task. If the object has been locked by a thread or the real-time task, the method is blocked until the object is unlocked and terminates the sharing after the object is unlocked. If the sharing has been terminated, a ShmIllegalStateException will be thrown.

public abstract void unshare(int timeout) throws ShmTimeoutException, ShmIllegalStateException
Terminates the object sharing with the real-time task. If the object has been locked by a different thread or the real-time task, the method is blocked for the *timeout* (unit ms), time-out time, until the object is unlocked. If the time-out time has elapsed, a **ShmTimeoutException** will be thrown. The method terminates the sharing after the object is unlocked. If the sharing has been terminated, a **ShmIllegalStateException** will be thrown.

public abstract Object getContent()

Returns the shared object. The **SharedObjectManager** actually obtains the shared object by using this method.

4.3.3 class jp.gr.itron.jtron.shared.SharedObject

```
java.lang.Object
   |
   +--- jp.gr.itron.jtron.shared.SharedObject
```

---

public class SharedObject
extends Object
implements Sharable

A shared object class. Programmers can easily create a shared object class by defining a subclass which inherits this class.

□ **Variables**

protected Sharable shm
has the object specified by the *shm*, the argument of a constructor. If a constructor which does not specify *shm* is called, this will be set.

□ **Constructor**

public SharedObject(String name) throws ShmIllegalStateException
Creates a shared object with *name*. The shared object is registered in the shared object manager when being created and the access to the object from the real-time task will be enabled. If the registration fails, a **ShmIllegalStateException** will be thrown.

public SharedObject(Sharable shm, String name) throws ShmIllegalStateException
Sets the *shm*, the object of the class implementing **Sharable**, as shared objects with *name*. The object will be registered in the manager when being created and the object can be referred to from the real-time task. If the registration fails, a **ShmIllegalStateException** will be thrown.

□ **Methods**

public void lock()
Locks the object. No action takes place if the object has been locked by the same thread. If the object has been locked by a different thread or the real-time task, the method is blocked until the object is unlocked.

public void lock(int timeout) throws ShmTimeoutException
Locks the object. No action takes place if the object has been locked by the same thread. If the object has been locked by a different thread or the real-time task, the method is blocked for the *timeout* (unit ms), *time-out* time, until the object is unlocked. If the *time-out* time has elapsed, a **ShmTimeoutException** will be thrown.

---

public vpid unlock() throws ShmIllegalStateException

Unlocks the object locked by the same thread. No action takes place if the object has been unlocked. Throws a **ShmIllegalStateException** for the object which has been locked by a different thread or the real-time task.

public void forceUnlock()

Unlocks the object which has been locked by a thread by force. No action takes place for the object being locked by the real-time task.

public void unshare() throws ShmIllegalStateException

Terminates the object sharing with the real-time task side. If the object has been locked by a thread or the real-time task, the method is blocked until the object is unlocked and terminates the sharing after the object is unlocked. If the sharing has been terminated, a **ShmIllegalStateException** will be thrown.

public void unshare(int timeout) throws ShmTimeoutException, ShmIllegalStateException

Terminates the object sharing with the real-time task. If the object has been locked by a different thread or the real-time task, the method blocked for the *timeout* (unit ms), until the object is unlocked. If the *time-out* time has elapsed, a **ShmTimeoutException** will be thrown. The method terminates the sharing after the object is unlocked. If the sharing has been terminated, a **ShmIllegalStateException** will be thrown.

public Object getContent()

Returns the shared object. The **SharedObjectManager** actually obtains the shared object by using this method. In the **SharedObject**, this method returns the **Sharable** object specified by the argument of a constructor (the implementation returns the instance variable shm).

A programmer can override this method if the programmer would like to have another object (such as array) as the sharing target in the **SharedObject** sub class. The below shows an example.

```
public class SharedData extends SharedObject {
    protected int data[];
    public SharedData(String name) {
        super(name);
        data = new int[10];
    }

    public Object getContent() {
        return data;
    }
    .....
}
```

4.3.4 Class jp.gr.itron.jtron.shared.SharedObjectManager

```
java.lang.Object
   |
   +--- jp.gr.itron.jtron.shared.SharedObjectManager
```

---

public abstract class SharedObjectManager

The class to manage the shared objects. This class is in charge of the interface with the real-time task. This class is an abstract class and vendors provide subclasses which inherit this class.

□ **Constructor**

protected SharedObjectManager()
Creates the shared object manager.

□ **Methods**

public static SharedObjectManager getSharedObjectManager() throws ShmIllegalStateException
Returns the manager object of the default shared object. If the manager cannot be provided or is illegal, a **ShmIllegalStateException** will be thrown.

public abstract void share(Sharable obj, String name) throws ShmIllegalStateException
Registers the *obj* under the name of *name*. If it has been registered or the name is illegal, a **ShmIllegalStateException** will be thrown.

public abstract void unshare(String name) throws ShmIllegalStateException
Deletes the object corresponding the *name*. If the object does not exist, a **ShmIllegalStateException** will be thrown.

public abstract void unshare(String name, int timeout) throws ShmIllegalStateException, ShmTimeoutException
Deletes the object corresponding the *name*. If the object does not exist, a **ShmIllegalStateException** will be thrown.

public abstract void lock(Sharable obj)
Locks the object. No action takes place if the object has been locked by the same thread. If the object has been locked by a different thread or the real-time task, the method is blocked until the object is unlocked.

public abstract void lock(Sharable obj, int timeout) throws ShmTimeoutException

Locks the object. No action takes place if the object has been locked by the same thread. If the object has been locked by a different thread or the real-time task, the method is blocked until the object is unlocked. Time-out time (unit: ms) can be specified and the method is blocked for the *timeout* time until the object is unlocked. If the *timeout* time has elapsed, a **ShmTimeoutException** will be thrown.

public abstract void unlock(Sharable obj) throws ShmIllegalStateException

Unlocks the object locked by the same thread. No action takes place if the object has been unlocked. Throws a **ShmIllegalStateException** for the object which has been locked by a different thread or the real-time task.

public absract void forceUnlock(Sharable obj)

Unlocks the object locked regardless of which thread locked the object. No action takes place if the object has been unlocked. The method is to be used when the thread which locked the object died without unlocking the object.

4.3.5 Class jp.gr.itron.jtron.shared.ShmException

```
java.lang.Object
   |
   +---- java.lang.Throwable
            |
            +---- java.lang.Exception
                     |
                     +---- jp.gr.itron.jtron.shared.ShmException
```

---

public class ShmException
extends Exception

Reports the occurrence of the exception related to shared objects. This class is a superclass of all the exception classes in this package.

□ **Constructor**

public ShmException()
Creates a **ShmException** without any detailed message.

public ShmException(String msg)
Creates a **ShmException** which has a specified detailed message, *msg*.

4.3.6 Class jp.gr.itron.jtron.shared.ShmIllegalStateException

```
java.lang.Object
   |
   +--- java.lang.Throwable
           |
           +--- java.lang.Exception
                   |
                   +---- jp.gr.itron.jtron.shared.ShmException
                           |
                           +---- jp.gr.itron.jtron.shared.ShmIllegalStateException
```

---

pubic class ShmIllegalStateException
extends ShmException

is thrown when a method is issued but the object is in illegal status disabling the execution of the method.

□ **Variables**

public static final int **ILLEGAL_MANAGER** = 1
Illegal shared object manager, or no manager exists.

public static final int **OBJECT_IN_USE** = 2
The object has been already registered.

public static final int **OBJECT_NOEXIST** = 3
The object does not exist anymore (already deleted).

public static final int **ILLEGAL_NAME** = 4
An illegal name.

public static final int **OBJECT_UNSHARED** = 5
The object has not been shared.

public static final int **OBJECT_LOCKED** = 6
The object has been locked by another thread or the task.

□ **Constructor**

public ShmIllegalStateException(int cause)
Creates the exception object for the specified cause. Gives the detailed cause of the exception to the parameter *cause*.

public ShmIllegalStateException(int cause, String msg)
Creates the exception object which has the detailed message of the specified cause. Normally, the detailed *cause* of the exception is specified to the cause and the object name is specified to the *msg*.

□ **Methods**

public int getCause()
Returns the detailed cause of the exception.

4.3.7 Class jp.gr.itron.jtron.shared.ShmTimeoutException

```
java.lang.Object
   |
   +--- java.lang.Throwable
           |
           +--- java.lang.Exception
                   |
                   +--- jp.gr.itron.jtron.shared.ShmException
                           |
                           +---- jp.gr.itron.jtron.shared.ShmTimeoutException
```

---

public class ShmTimeoutException
extends ShmException

Reports that the time-out time has elapsed.

□ **Constructor**

public ShmException()
Creates a **ShmTimeoutException** without any detailed message.

public ShmTimeoutException(String msg)
Creates a **ShmTimeoutException** which has a specified detailed message, *msg*. Normally, the object name is specified to the *msg*.

Chapter 5

STREAM INTERFACE

5.1 General

5.1.1 What is stream interface

The stream interface provides the communication between the real-time task and the Java thread by using the **InputStream** and **OutputStream** classes which are the standard input/output interfaces in Java APIs.

In the Java program, the stream which communicates with the real-time task is provided as implementation of the abstract class **InputStream** and **OutputStream**. This is similar to the abstract class implementation of the **InputStream** and **OutputStream** classes from the **Socket** class.

On the ITRON task, the mechanism which performs the stream communication with the Java program is provided as parts of the RTOS.

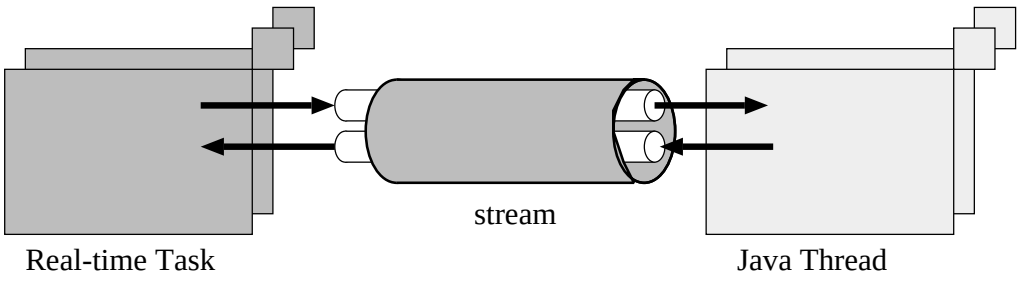


Figure 5.1: Streams

5.1.2 Stream and channel status

The streams are identified by the identification numbers.

The resource management, such as a buffer for stream communication, is performed on the RTOS. The creation and the deletion of streams are done from the real-time task (**jti_cre_stm**/**jti_del_stm**).

A stream consists of two channels; a channel to send data from the real-time task to the Java program and a channel to send data from the Java program to the real-time task. By specifying a parameter in the creation of the stream, a stream which has only one of the channels can be also created.

The created stream is in the UNCONNECTED status. For the streams in the UNCONNECTED status, the programmer can open the stream in the Java program (**JtiDataStream**) with the stream identification number.

When a stream is opened, both of the channels are connected (in case of one-channel stream, only that channel will be connected. The other channel is considered to have been disconnected.).

When the sending program of the channel closes the channel normally (**close** for the **outputStream** from the Java program and **jti_sht_stm** from the real-time task), the channel will be in the CLOSED status. After the receiving program of the channel takes the data from the buffer and detects the normal closing (the normal closing is detected by the return of -1 from the **inputStream.read** in the Java program, and 0 from the **jti_rea_stm** in the real-time task), the channel will be disconnected at the point when the closing is confirmed (confirmed by the **close** of the **inputStream** in the Java program, and by the return of 0 from the **jti_rea_stm** in the real-time task). When both of the channels are disconnected, the stream goes back to the UNCONNECTED status.

The channel used for receiving as **inputStream** and the channel used for sending as **outputStream** in the Java program . If the Java program closes the channel of the receiving side by force (**close** of the **inputStream**), the channel will be in the FORCED DISCONNECTED status. When the real-time task, the sending side of the channel, detects and confirms the closing (real-time task regards the return of **E_CLS** from the **jti_wri_stm** or **jti_sht_stm** as the detection and confirmation of the enforced closing), the channel will be disconnected. The real-time task cannot force to close the channel of which the task is the receiving side(no API in the real-time kernel provided for the enforced closing).

The **close** of the **JtiDataStream** is equivalent to the **close** of both of the **outputStream** and the **inputStream**.

[**Supplementary explanation**]

When the **jti_rea_stm** returns 0 (or **jti_wri_stm** or **jti_sht_stm** returns the **E_CLS**), it is regarded that the real-time task confirmed the normal closing (or enforced closing) and the channel status changes. Since 0 (or **E_CLS**) will not be returned even if the real-time task call the **jti_rea_stm** (or **jti_sht_stm** or **jti_wri_stm**) again, a programmer should pay attention.

The stream status takes a status of "NON-EXISTENT" or "UNCONNECTED" or one of the other 11 status depending on the status of both channels. In detail, there are 3 status, "CONNECTED", "CLOSED", and "CONNECTED" in the channels from Java program to the real-time task, and 4 status, "CONNECTED", "CLOSED","FORCED DISCONNECTED", and "CONNECTED" in the channels from the real-time task to the Java program. In addition, the stream goes back to the UNCONNECTED status when both channels are

disconnected. Therefore, in total, stream can take one of 13 status ($= 2 + 3 \times 4 - 1$).

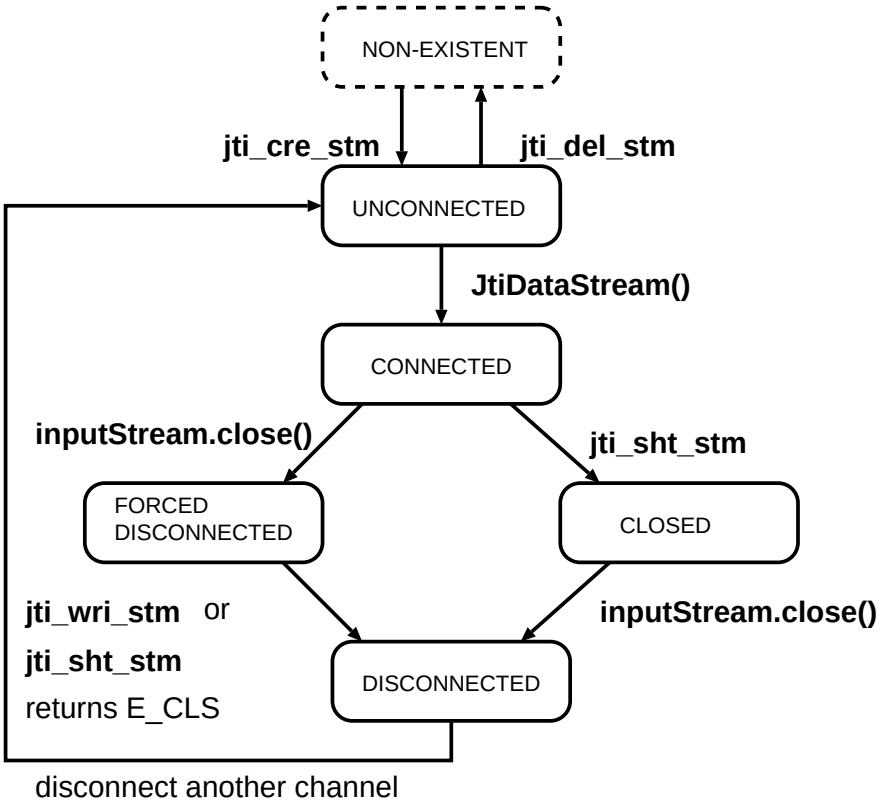


Figure 5.2: Channel status transition from real-time task to Java program

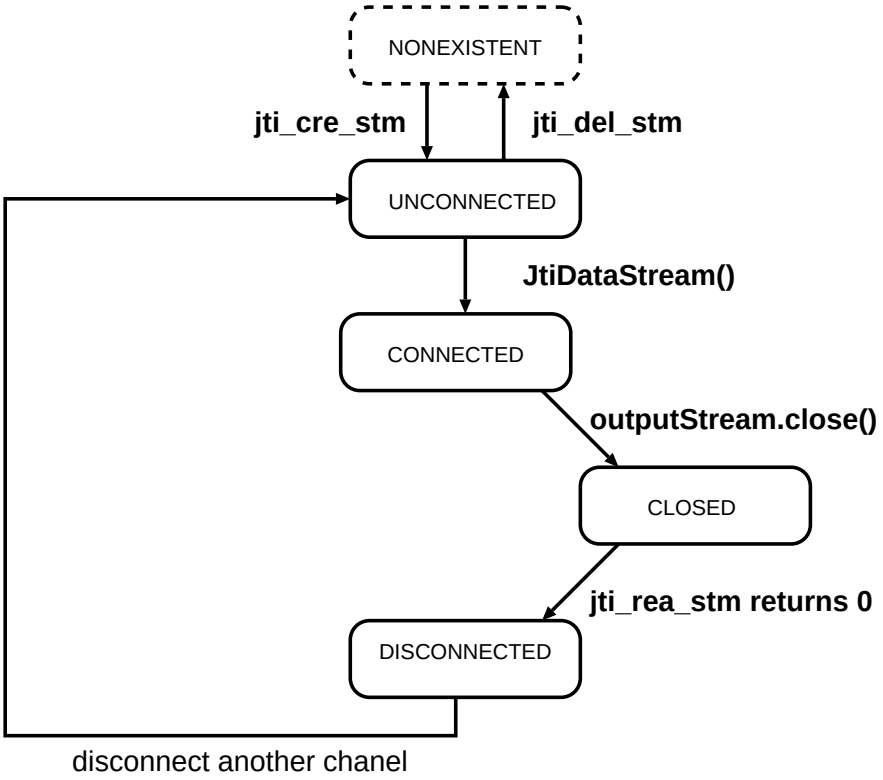


Figure 5.3: Channel status transition from Java program to real-time task

5.2 ITRON API

5.2.1 Creating / Deleting streams

API Name	Function	Type
jti_cre_stm, JTI_CRE_STM	Creates streams	Standard
jti_del_stm	Deletes streams	Standard

Standard

jti_cre_stm, JTI_CRE_STM

Creates streams

[**C language API**]

```
ER ercd = jti_cre_stm(ID stmid, T_JTI_CSTM *pk_cstm);
```

[**Static API**]

```
JTI_CRE_STM(ID stmid,
            { VP exinf, ATR stmpatr, VP wbuf,
              INT wbufsz, VP rbuf, INT rbufsz });
```

[**Parameters**]

ID	*stmid*	Stream identifier
T_JTI_CSTM	**pk_cstm*	Stream creation information

Contents of pk_cstm

VP	exinf	Extension information
ATR	stmatr	Stream attribute
VP	wbuf	Head of the sending buffer
INT	wbufsz	Sending buffer size
VP	rbuf	Head of the receiving buffer
INT	rbufsz	Receiving buffer size

(Other implementation-dependent parameters are also acceptable)

[**Return value**]

ER	*ercd*	Error code

[**Error codes**]

E_OK	Normal termination
E_ID	Illegal ID number
E_RSATR	Reserved attribute
E_PAR	Parameter error (Illegal *pk_cstm* address, wbuf, wbufsz, rbuf, rbufsz, stream attribute)
E_OBJ	Object status error (The stream with the specified identifier already being created)

[API function]

Creates the stream with the specified identifier. By using the stream attribute, the stream can be exclusive for sending or receiving. Specifying the (**TA_WRITE**|**TA_READ**) to the stream attribute enables the bilateral communication and specifying **TA_WRITE** and **TA_READ** make the stream exclusive for sending and for receiving, respectively. If a programmer specifies the stream attribute other than **TA_WRITE** or **TA_READ**, an **E_RSATR** will be returned. If the programmer does not specify either of **TA_WRITE** and **TA_READ**, an **E_PAR** error will be returned.

When the stream is exclusive for sending, *rbuf* and *rbufsz* will be ignored. If the stream is exclusive for receiving, *wbuf* and *wbufsz* will be ignored. If the sending/receiving buffer size is negative, an **E_PAR** error will be returned (asynchronous communication in case of the buffer size 0).

In the implementation where the buffer is allocated inside, **NADR**($= -1$) should be specified as a head address of the buffer(*wbuf*, *rbuf*). In this case, specifying the buffer size is valid. It is also allowed to have the implementation which allocates the buffer inside in case of specifying **NADR** and uses the given buffer in the other cases.

Standard

jti_del_stm

Deletes streams

[**C language API**]

```
ER ercd = jti_del_stm(ID stmid);
```

[**Parameters**]

ID *stmid* Stream identifier

[**Return value**]

ER *ercd* Error code

[**Error codes**]

E_OK	Normal termination
E_ID	Illegal ID number
E_NOEXS	Object not created
E_OBJ	Object status error (The specified stream is not in the UNCONNECTED status)

[**API function**]

Deletes the specified stream. If the program tried to delete the stream other than in the UNCONNECTED status, an **E_OBJ** error will be returned. Tasks in the waiting status after issuing the **jti_rea_stm** or **jti_wri_stm** will be activated and return the ECDE_DLT.

5.2.2 Sending/Receiving data and ending the sending

API Name	Function	Type
jti_wri_stm	Sends data	Standard
jti_rea_stm	Received data	Standard
jti_sht_stm	Ends the data sending	Standard

Standard

jti_wri_stm

Sends data

[**C language API**]

```
ER ercd = jti_wri_stm(ID stmid, VP data, INT len, TMO tmout);
```

[**Parameters**]

ID	*stmid*	Stream identifier
VP	*data*	Head address of the data to be sent
INT	*len*	Length of the data to be sent
TMO	*tmout*	Time-out time

[**Return value**]

ER	*ercd*	Length of the data in the buffer / Error code

[**Error codes**]

Positive value	Normal termination (Length of the data in the sending buffer)
E_ID	Illegal ID number
E_NOEXS	Object not created
E_PAR	Parameter error (illegal *data*,*len*,*tmout*)
E_OBJ	Object status error (The specified stream is exclusive for receiving, **jti_wri_stm** is pending), the stream waiting in the UNCONNECTED status was deleted
E_TMOUT	Polling failure or time-out
E_RLWAI	Compulsory release of the waiting status
E_CLS	The channel for sending was disconnected by force

[**API function**]

Sends data to the specified stream and returns from this API when the data is entered to the sending buffer. If the sending buffer length is shorter than a length(*len*) of the data to be sent, the data is entered in the sending buffer until the sending buffer becomes full and the length of the data entered in the sending buffer is returned. If there is no space in the sending buffer, the API will be in the waiting status until the buffer is available.

The data sending to the stream is accepted only when the channel for sending is in the connected status. If the channel is in the FORCED DISCONNECTED status, **jti_wri_stm** returns an **E_CLS** error and

the channel transfers to the DISCONNECTED status. In any other status (DISCONNECTED, UNCONNECTED), the task which called the **jti_wri_stm** will be in the waiting status until the channel changes to the CONNECTED status.

If a **jti_wri_stm** is issued while the **jti_wri_stm** for the same stream is pending, an **E_OBJ** error will be returned.

Standard

jti_rea_stm

Receives data

[**C language API**]

```
ER ercd = jti_rea_stm(ID stmid, VP data, INT len, TMO tmout);
```

ID	*stmid*	Stream identifier
VP	*data*	Head address of the region to put the data received
INT	*len*	Length of the data to be received
TMO	*tmout*	Time-out specification

[**Return value**]

ER	*ercd*	Length of the data received / Error code

[**Error codes**]

Positive value	Normal termination (length of the data taken out)
0	End of data (connection was normally disconnected)
E_ID	Illegal ID number
E_NOEXS	Object not created
E_PAR	Parameter error (illegal *data*, *len*, *tmout*)
E_OBJ	Object status error (The specified stream is exclusive for sending, **jti_rea_stm** is pending), the stream waiting in the UNCONNECTED status was deleted
E_TMOUT	Polling failure or time-out
E_RLWAI	Compulsory release of the waiting status

[**API function**]

Receives data from the specified stream and returns from this API when the data put in the receiving buffer was read. If the data length in the receiving buffer is shorter than the specified data length(*len*) to be received, the data will be read until the receiving buffer becomes empty and the length of the data read will be returned. If the receiving buffer is empty, the API will be in the waiting status until any data arrives. When the Java program closes the receiving channel and no data is left in the receiving buffer, 0 will return from the API.

The data receiving from the stream is accepted only when the channel for receiving is in the CONNECTED status. If the channel is in the other status (DISCONNECTED, UNCONNECTED), the task which called the **jti_rea_stm** will be in the waiting status until the channel changes to the CONNECTED status.

If a **jti_rea_stm** is issued while the **jti_rea_stm** for the same stream is pending, an **E_OBJ** error will be returned.

Standard

jti_sht_stm

Ends the data sending

[**C language API**]

```
ER ercd = jti_sht_stm(ID stmid);
```

[**Parameters**]

ID *stmid* Stream identifier

[**Return value**]

ER *ercd* Error code

[**Error codes**]

E_OK	Normal termination
E_ID	Illegal ID number
E_NOEXS	Object not created
E_OBJ	Object status error (The specified stream is exclusive for receiving, the channel for sending is in the DISCONNECTED status or UNCONNECTED status, **jti_wri_stm** is pending)
E_CLS	The channel for sending was disconnected by force

[**API function**]

Ends the data sending to the specified stream and transfers the sending channel to CLOSED status.

The data sending to the stream is accepted only when the channel for sending is in the CONNECTED status. If the channel is in the FORCED DISCONNECTED status, **jti_sht_stm** returns an **E_CLS** error and the channel transfers to the DISCONNECTED status. In any other status (DISCONNECTED, UNCONNECTED), an **E_OBJ** error will be returned.

If a **jti_sht_stm** is returned while the **jti_wri_stm** for the same stream is pending, an **E_OBJ** error will be returned.

5.2.3 Refers to the stream status

API Name	Function	Type
jti_ref_stm	Refers to the stream status	Standard

Standard

jti_ref_stm

Refers to the stream status

[**C language API**]

```
ER ercd = jti_ref_stm(ID stmid, T_JTI_RSTM *pk_rstm);
```

[**Parameters**]

ID	*stmid*	Stream identifier
T_JTI_RSTM	**pk_rstm*	Address of the packet to return the stream status

[**Return value**]

ER	*ercd*	Error code

Contents of pk_rstm

VP	exinf	Extension information
INT	wrisz	Data length which can be sent without waiting (Number of bytes)
INT	reasz	Data length which can be received without waiting (Number of bytes)

(Other implementation-dependent parameters can also be added)

[**Error codes**]

E_OK	Normal termination
E_ID	Illegal ID number
E_NOEXS	Object not created
E_PAR	Parameter error (illegal *pk_rstm* address)

[**API function**]

Refers to the specified stream status and returns the status to the *pk_rstm*.

The extension information specified in the **jti_cre_stm** will return to *exinf*. The data length which can be sent without waiting (number of bytes) will return to *wrisz*. The data length which can be received without waiting (number of bytes) will return to *reasz*. If the specified stream is exclusive for sending, -1 will be returned.

5.3 Java API

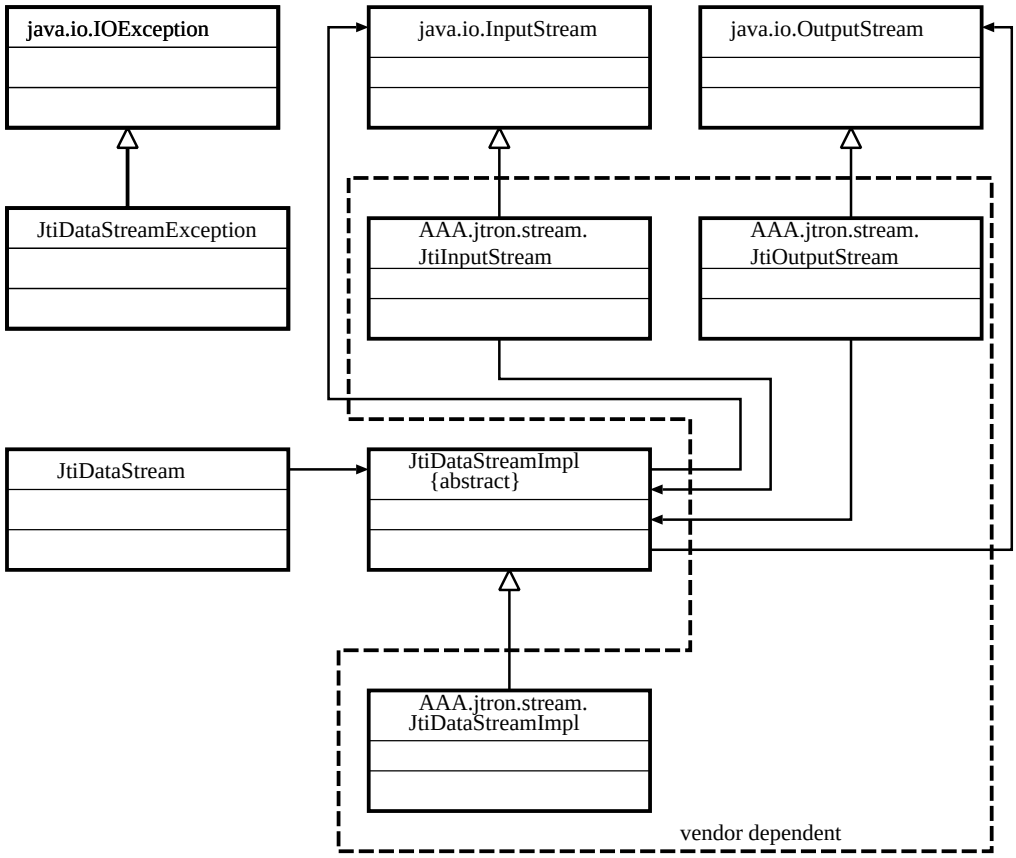
5.3.1 Package structure

The classes providing the streams are put together in the jp.gr.itron.jtron.stream package. This package consists of the following class and exception class.

Class: JtiDataStream

Exception class: JtiDataStreamException

The stream package class structure is shown in the figure 5.4. The **JtiDataStream** and its implementation, **JtiDataStreamImpl**, are separated by using a design pattern called bridge [5][6] in implementing the **JtiDataStream**. This realizes to exchange easily the different stream implementation by vendors. The bridge is also used in the **java.io.Socket**.



The relation between the JtiDataStreamImpl and the stream classes is implementation-dependent. The relation shown above is one of the implementation example. There may be a relation between the JtiDataStreamImpl subclass (AAA.jtron.stream.JtiDataStreamImpl in the above figure) and the stream classes in a certain implementation.

Figure 5.4: Stream package class structure

5.3.2 Class jp.gr.itron.jtron.stream.JtiDataStream

```
java.lang.Object
   |
   +--- jp.gr.itron.jtron.stream.JtiDataStream
```

public class JtiDataStream

The class to communicate with the real-time task by using streams.

□ **Variables**

public static final int **MAIN_STREAM** = 1
A standard stream identifier used between the real-time task and the Java program.

□ **Constructor**

public JtiDataStream(int stmid)throws JtiDataStreamException
Opens the stream with the specified identifier. This sets both channels into the CONNECTED status (if the stream has only one channel, only that channel will be connected). If the specified identifier is already in use, a **JtiDataStreamException** will be thrown.

public JtiDataStream(int stmid, int timeout)throws IOException,InterruptedException
Opens the stream with the specified identifier and *timeout* time. The unit of the *timeout* time is millisecond. If the specified identifier is already in use, a **JtiDataStreamException** will be thrown. When the *timeout* time has elapsed, an **InterruptedException** will be thrown.

protected JtiDataStream(JtiDataStreamImpl impl, int stmid, int timeout) throws IOException, InterruptedException
Opens the stream with the specified identifier with the *timeout* time by using the user-defined implementation. The unit of the *timeout* time is millisecond. If the specified identifier is already in use, a **JtiDataStreamException** will be thrown. When the *timeout* time has elapsed, an **InterruptedException** will be thrown.

□ **Methods**

public synchronized InputStream getInputStream() throws IOException
Obtains the receiving stream.

public synchronized OutputStream getOutputStream() throws IOException
Obtains the sending stream.

public synchronized void setIDSTimeOut(int timeout) throws IOException

Sets the *timeout* time for the case the **read** method for the **InputStream** is executed. The unit of the *timeout* time is millisecond and the *timeout* must be 0 or bigger. If 0 has been specified for the *timeout*, it will cause the permanent waiting. When the *timeout* time has elapsed, a **java.io.InterruptedException** will be thrown. However, if the **write** method was executed for the **OutputStream**, specifying the time-out will not be available just as the cases for the other stream operations.

public synchronized int getIDSTimeOut() throws IOException

Obtains the time-out time for the case the **read** method for the **InputStream** is executed. The unit of *timeout* time is millisecond. Return of 0 means the permanent waiting.

public synchronized void close() throws IOException

Closes the stream. If the sending channel has been connected, the method normally closes the stream to change the status to CLOSED status, and if the receiving channel has been connected, this method closes the stream by force to change the status to the FORCED DISCONNECTED status. This method also puts the status to the DISCONNECTED status if the receiving channel is in the CLOSED status

5.3.3 Class jp.gr.itron.jtron.stream.JtiDataStreamImpl

```
java.lang.Object
   |
   +--- jp.gr.itron.jtron.stream.JtiDataStreamImpl
```

---

public abstract class JtiDataStreamImpl

An abstract class to define classes which have a stream interface implementation. This class is provided to separate the specification and the implementation.

□ **Constructor**

public JtiDataStreamImpl() throws JtiDataStreamException

□ **Methods**

Those without explanation have the same specification as the corresponding methods in the jp.gr. itron.jtron.stream.JtiDataStream.

public abstract void setTimeout(int timeout)
Sets the time-out time.

public abstract void setStreamId(int stmid)
Sets the identifier.

public abstract int getTimeout(int timeout)
Obtains the time-out time.

public abstract int getStreamId(int stmid)
Obtains the identifier.

public abstract InputStream getInputStream() throws IOException

public abstract OutputStream getOutputStream() throws IOException

public abstract void setIDSTimeOut(int timeout) throws IOException

public abstract int getIDSTimeOut() throws IOException

public abstract void close() throws IOException

5.3.4 Class jp.gr.itron.jtron.stream.JtiDataStreamException

```
java.lang.Object
   |
   +--- java.lang.Throwable
           |
           +--- java.lang.Exception
                   |
                   +--- java.io.IOException
                           |
                           +--- jp.gr.itron.jtron.stream.JtiDataStreamException
```

public class JtiDataStreamException extends IOException

Reports the occurrence of the exception related to the stream communication.

□ **Constructor**

public JtiDataStreamException(int cause)
Creates the **JtiDataStreamException** without any detailed message. Gives the detailed cause of the exception to the parameter *cause*.

public JtiDataStreamException(int cause, String msg)
Creates the **JtiDataStreamException** with has the specified detailed message, *msg*. Gives the detailed cause of the exception to the parameter *cause*.

□ **Methods**

public int getCause()
Returns the detailed cause of the exception.

□ **Variables**

public static final int **STREAM_NOT_FOUND** = 1
No stream with the specified identifier exists (not created).

public static final int **STREAM_IN_USE** = 2
The stream with the specified identifier is already in use.

public static final int **STREAM_CLOSED** = 3
The stream with the specified identifier has been already closed.

public static final int **STREAM_ILLEGAL_ARGUMENT** = 4
The specified argument is illegal.

Appendix A

APPENDIX

A.1 Attach Classes

TBD

A.2 Shared Object Interface

A.2.1 Definition examples

(1) When inheriting

If it is not a subclass of other classes, it is recommended to inherit the SharedObject class.

- Example 1

```
public class MyObject extends SharedObject {
    private int x;
    private int y;
    private int z;

    public MyObject(String name) {
      super(name);
      ...
    }
    ....
}
```

- Example 2

```
public class SharedData extends SharedObject {
    int data[];
    ....
    public SharedData(String name) {
      super(name);
      data = new int[10];
      ...
    }
    public Object getContent() {
        return data;
    }
    ....
}
```

(2) When using Sharable interface

Have the SharedObject class as a member.

```
public class FooObject implements Sharable {
    private SharedObject shm;
    int x;
    int y;
    int z;

    public FooObject(String name) {
        shm = new SharedObject(this, name);
    }
    public void lock() {
        shm.lock();
    }
    public void lock(int timeout) {
        shm.locj(timeout);
    }
    public void unlock() {
        shm.unlock();
    }
    public void forceUnlock() {
        stm.forceUnlock();
    }
    public void unshare() {
        shm.unshare();
    }
    public void unshare(int timeout) {
        shm.unshare(timeout);
    }
    public void getContent() {
      return this;
    }
}
```

A.2.2 Communication examples by real-time task and Java program

Java side(JtiSharedSample.java)

List A.1 JtiSharedSample.java

```
1   /**
2    * Shared Object sample
3    */
4   import jp.gr.itron.jtron.shared.*;
5
6   class SharedData extends SharedObject {
7       private int data;
8
9       SharedData(String name) throws ShmIllegalStateException {
10          super(name);
11          data = 0;
12      }
13
14      public int getData() {
15          return data;
16      }
17  }
18
19  public
20  class JtiSharedSample {
21      private SharedData data = null;
22      private int sum;
23
24      JtiSharedSample() {
25          sum = 0;
26          try {
27              data = new SharedData("Shared");
28          } catch (ShmIllegalStateException ex) {
29              System.out.println("error: code =" + ex.getCause());
30              System.exit(1);
31          } catch (ShmException ex) {
32              System.out.println("error:" + ex);
33              System.exit(1);
34          }
35      }
36
37      public void dispose() {
38          try {
39              data.unshare(); // Executes unshare
```

```
40          } catch (ShmIllegalStateException ex) {
41              System.out.println("error: already unshared.");
42          }
43      }
44
45      public void startSharedSample() {
46          int c;
47
48          try {
49              while(true) {
50                  c = 0;
51                  try {
52                      data.lock(10);      // Waits the data from the real-time task
53                      c = data.getData(); // Obtains the data
54                      data.unlock();
55                      if (c == -1) {      // Ends when -1 was sent.
56                          break;
57                      }
58                      sum += c;           // Processes the data
59                  } catch (ShmTimeoutException ex) {
60                      /* Process for the timeout: Sleeps for 10ms here */
61                      try {
62                          Thread.sleep(10);
63                      } catch (InterruptedException e) {
64                          /* nop */
65                      }
66                  }
67              }
68              System.out.println("sum      = " + sum);
69          } catch (ShmIllegalStateException ex) {
70              System.out.println("internal error:" + ex);
71          }
72      }
73
74      public static void main(String args[]) {
75          JtiSharedSample app = new JtiSharedSample();
76          app.startSharedSample();
77          app.dispose();
78      }
79  }
```

Appendix APPENDIX

ITRON side (jtron.c)

List A.2 jtron.c

```
1   #include "jti_shared.h"
2   #define  WAIT_TIME  10 /* Maximum lock waiting time */
3
4   struct JSharedObj *p;  /* Structure by javah of the shared object */
5
6   void maintask() {
7       JNO  shoid;
8       ER  ercd;
9       /* Obtains the shared object id with the jti_get_obj by using the name */
10      ercd = jti_get_obj("Shared", &shoid);
11
12      while(1) {
13          /* Intends to lock the shared object with the jti_loc_obj. Waiting time is 10 milliseconds */
14          ercd = jti_loc_obj(shoid, MAX_TIME);
15          if (ercd == E_OK) { /* When the object could be locked */
16              /* Obtains the address of the shared object with the jti_get_mem */
17              jti_get_mem(&p, shoid);
18              /* Prepares the data to be given to the Java program side */
19              /* Sets the data to the region specified by the p */
20
21              /* Unlocks the shared object with the jti_unl_obj */
22              ercd = jti_unl_obj(shoid);
23          }
24      }
25  }
```

A.3 Stream Interface

A.3.1 Communication examples by real-time task and Java program

ITRON side (jtron.c)

List A.3 jtron.c

```
1   #include "jti_stream.h"
2   #define SIZE_WBUF 100
3   char WBUF[SIZE_WBUF];
4
5   #define N_DATA 100
6
7   /* ITRON => Java */
8
9   void maintask() {
10      ER ercd;
11      T_JTI_CSTM pk_cstm;
12      int writedata, readdata;
13      int i;
14
15      /* Creates a stream (Write side only) */
16      pk_cstm.exinf = 0;
17      pk_cstm.stmatr = TA_WRITE;
18      pk_cstm.wbuf = WBUF;
19      pk_cstm.wbufsz = SIZE_WBUF;
20      pk_cstm.rbuf = 0;
21      pk_cstm.rbufsz = 0;
22
23      ercd = jti_cre_stm(JTI_MAIN_STREAM, &pk_ctsm);
24
25      /* Sends the data */
26      /* Enters the waiting status until the ItronDataStream() is called on the Java side */
27      /* Sends the N_DATA units of data from the ITRON side */
28
29      for (i = 0; i < N_DATA; i++) {
30          writedata = i;
31          ercd = jti_wri_stm(JTI_MAIN_STREAM, &writedata, sizeof(int), TMO_FEVR);
32          /* What to do in case of error? */
33          if (ercd != E_OK) {
34              /* Processes the error */
35          }
36      }
37      /* Ends the data sending */
```

```
38      ercd = jti_sht_stm(JTI_MAIN_STREAM);
39      /* Deletes the stream */
40      ercd = jti_del_stm(JTI_MAIN_STREAM);
41
42      ext_tsk();
43  }
```

Java side (JtiStreamSample.java)

List A.4 JtiStreamSample.java

```
1   import java.net.*;
2   import java.io.*;
3   import java.util.*;
4
5   import jp.gr.itron.jtron.*;
6
7   public class JtiStreamSample {
8       public static void main(String args[]) {
9           JtiSteamSample jtiss = new JtiStreamSample();
10          jtiss.startStreamSample();
11      }
12
13      public void startStreamSample() {
14          JtiDataStream ids = null;
15          InputStream is = null;
16          int c = 0;
17
18          try {
19              // Connects with the ITRON side
20              ids = new JtiDataStream(MAIN_STREAM);
21
22              // Obtains the InputStream
23              is = ids.getInputStream();
24              while (true) {
25                  // Obtains the data sent from the ITRON side
26                  c = is.read();
27
28                  if (c == -1) {
29                      // Closes and terminates the process if the EOF was sent
30                      is.close();
31                      return;
32                  }
33                  // Process for the data read
34              }
35              is.close();
36              return;
37          } catch (IOException ioe) {
38              System.out.println("Exception:" + ioe.getMessage());
39              return;
40          }
41      }
```

```
42  }
```

Index